

Aspects in the Agile Toolbox¹

Gustav Boström

gusbo@kth.se

Phone: +46-8-752 17 26

Fax: +46-8-703 90 25

Martin Henkel

martinh@dsv.su.se

Phone: +46-8-16 16 42

Fax: +46-8-703 90 25

Jaana Wäyrynen

jaana@dsv.su.se

Phone: +46-8-674 74 67

Fax: +46-8-703 90 25

Department of Computer and Systems Science

Stockholm University/Royal Institute of Technology

Forum 100, SE-164 40 Kista

Sweden

¹ The names of the authors appear in alphabetical order

Aspects in the Agile Toolbox

Abstract

The introduction of agile methodologies promises faster time to market and a flexible, customer-driven software development process. The flexibility with regards to requirements can however also be a risk if new non-functional requirements need to be implemented late in a software development project. In this article we argue that aspect-oriented programming is a technique that can mitigate this risk by providing the means to deal with non-functional, crosscutting requirements. The suggested solution is illustrated by an example where new requirements in the form of quality of service demands on an existing service are implemented by using aspect orientated programming.

Introduction

In recent years, a new style of software development has been introduced to the field. Driven by rapidly changing business needs, agile software development methods have been proposed to overcome the problems experienced with more traditional, heavyweight software development methods. Agile methods are based on the enforcement of the practices of iterative, incremental and adaptive development, where simplicity is one of the core values. Reduced to nothing but the essential (Kelter et al., 2002), agility is a technique that responds to the need to develop information systems more quickly in the competitive business environment of today.

The growing interest in the agile methodology shows that the idea of rationalising traditional software development can be fruitful, but the field is new and research is still in an initial state. One question that repeatedly has been illuminated, by both practitioners as well as academics, is the applicability of the agile emphasis on simplicity. YAGNI (You Aren't Going to Need It) (Beck, 2000), a philosophy of the agile method of eXtreme Programming (XP), symbolises the idea of simplicity as it emphasises getting rid of architectural features that are not needed in the current system version (Lindvall et al., 2002). Boehm is among those who argue that this approach of "simplicity" fits in situations when the requirements are unknown. In cases where future requirements are known, the risk is, however, that the lack of initial architectural support could cause severe architectural problems later (Lindvall et al., 2002).

Agile methods do not promote implementation of vague requirements that originates from guesses. The reason is to avoid an up-front design that hinders swift changes in the implementation and design later. In a way, agile methods are postponing the design and implementation of requirements until they are needed. This is fine for requirements that affect a small part of a system, since these changes can be accounted for by redesigning a minor

part of the system. However, using this approach of just-in-time design makes agile projects vulnerable to requirement changes that affect the entire, or a large part of a system. Systems built with an agile mindset have no extra “built-in” buffer for dealing with anticipated future requirements that a system built with an up-front design might have. This problem is especially pronounced when changing non-functional requirements, as these tend to have an impact on the entire system. In this respect, agile methods have an Achilles heel in handling cross-cutting, non-functional requirements that are discovered late in the development project. The question is how to overcome this weakness.

Aspect Oriented Programming (AOP) is proposed as a technique to implement functionality that cuts across an entire system (Duclos, 2002). Kircher et al. (2002) have also described possible positive effects of aspect oriented programming on the agile method eXtreme Programming. Thus, combining agile methods with aspect orientation may be a solution that decreases the risk of exposure to changing non-functional requirements, without losing the agility that are the trademarks of agile processes.

In this article we argue that aspect oriented programming can be a useful tool for agile projects that encounter new non-functional requirements that crosscut the application. The article begins with a short introduction to agile methods and aspect oriented programming. The introduction is followed by a description of the problem and possible solutions. Then, we provide an example of how aspect-orientation can be applied in the context of a service that needs to be changed due to changed non-functional requirements. The article ends with a discussion of the proposed solution’s applicability and a discussion of further work.

Agile Software Development

Facing the growing complexity of the business environment, traditional information systems development methods are challenged by the introduction of new styles of information systems development. In the 1990's so-called lightweight or agile software development methods have been proposed to respond to the need to create and amend information systems more speedily.

A number of software development methods belong to the agile family, where the most known methods are eXtreme Programming (XP), Scrum, Feature Driven Development (FDD), and Dynamic Systems Development Method (DSDM). Following the principles of the Manifesto for Agile Software Development (Beck et al., 2001), these are all a reaction to traditional development practices, making a compromise between no defined process and strictly defined process in the development of software. Agile methods are proposed to be used if flexibility, adaptability, and changing requirements are the main characteristics of a project – and in contrast to traditional methods, if strict control of progress and products is of minor importance (Kelter et al., 2002), (Lindvall et al., 2002), (Wendorff, 2002).

As stated earlier, the use of agile methods does not promote implementation of vague requirements that originate from guesses. Part of the reason is that, known to the community of software developers, the management of requirements in software development projects can be a cumbersome activity, for example due to the customers' inability to define long-term requirements up-front, but also due to external and environmental changes impacting the requirements initially specified. These experiences indicate that information systems development in most organisations is unable to react quickly enough, and the business and systems development cycles are therefore substantially out of step. To overcome these problems, agile methods are stressing the value of just-in-time design by expressing it as

principles such as designing “the simplest thing that could possibly work” (Beck, 2000), and avoiding “big design up front” (BDUF)(Lindvall et al., 2002).

Aspect Oriented Programming

Aspect oriented programming is a paradigm that attempts to help in implementing concerns that are present in many modules and therefore crosscuts a system, that is *cross-cutting concerns*. Cross-cutting concerns are difficult to modularise using existing object-oriented techniques since there is no logical place in which to implement them. An illustrative example is logging of method-calls. Using existing object-oriented techniques the code for implementing this would be spread out in all methods that require logging. Changing the way logging is done, and especially, where it is performed is therefore difficult to accomplish without changing all methods that need to be logged. This poor modularization leads to code that is difficult to maintain. The fact that you also need to deal with several concerns, logging and business logic, in the same method also adds to the complexity of the code.

Aspect oriented programming provides a way to modularise these cross-cutting concerns in an efficient manner by factoring out logic belonging to a specific concern into an *Aspect* (Elrad et al., 2001). In this article we use AspectJ as a tool to implement aspect oriented programming (AspectJ). An aspect in AspectJ consists of *join points*, and an *advice* (in AspectJ an aspect can also contain *Introduction*, but this concept is not used in this article). Join points describe *where* the aspect should apply in terms of the object-oriented systems structure, e.g. the join point of the logging aspect would describe *where* in the system logging should be performed in terms of the classes and methods of the system. An advice describes *what* should happen at these join points, e.g. how the logging should be carried out.

The AspectJ keyword *pointcut* is used to define a join point. The process of combining the aspects and the classes into an executable system is called *aspect weaving*.

Functional, non-functional and cross-cutting requirements

Functional requirements are statements of services that the system should provide. This can also be said as describing *what* the system should do. Non-functional requirements, on the other hand, are focused on *how* the system should perform the services (Cysneiros et al., 2002). An example of a functional requirement on an ATM-machine could for example be that an ATM-machine should be able to dispense money to the bank's customers. A non-functional requirement could be that this service has to be performed *securely* and with an acceptable *response time*. Other examples of non-functional requirements are performance, traceability, scalability and error handling. A problem with non-functional requirements is that they are often *crosscutting*, i.e. they affect many modules of the system. For example, security needs to be addressed in many parts of an ATM-system. It is therefore difficult to modularise crosscutting requirements. This can make systems difficult to maintain and evolve (Moreira et al., 2002).

The Achilles heel of agile methods: non-functional requirements

Living in an unpredictable world one can assume that requirements will change. If they change, how would new non-functional requirements that may crosscut an entire application in an agile development process be handled? Refactoring would be the normal answer, but then several other questions rises, such as "How well does that work when two hundred methods needs to be rewritten?", "Is the cost of change really low in this case?" Existing refactoring techniques (Fowler, 1999) mostly assume that the changes are local to one class or a few classes. This is not the case when requirements crosscut big parts of the system.

Security and performance are examples of areas in which new requirements can easily spread to large areas of an application. These kinds of crosscutting requirements normally require that you have built in these mechanisms beforehand in the system's architecture, but this is directly contrary to the idea in XP of simple design, which states that you should *not* build for the future (Jeffries, 2001). Barry Boehm (2002) argues that this is a reason to be careful when using agile software development methods on projects in which future requirements can in fact be guessed up-front.

To summarise, agile methods have an Achilles heel in handling new non-functional requirements that are discovered late in the development project. There are three obvious solutions to this situation:

- a) Abandoning the agile methods
- b) Ignore the risk and keep using the agile methods
- c) Augmenting agile methods with techniques that makes it easier to implement changes in non-functional requirements late in the project.

Clearly (a), avoiding agile methods is not an option if one is interested in the benefits of agile methods, such as fast time to market. The risk of using agile methods in environments where the non-functional requirements can change (b) can be compensated by saving resources as a buffer that can be used later, if the non-functional requirements change. However, most organisations will tend to pay for what they get, and not pay for something they might eventually need. The third option (c) of using agile methods in conjunction with techniques that lessen the impact of non-functional requirements is clearly attractive, as it gives the opportunity to take advantage of the benefits of agile methods, and at the same time lessen the risk of major rework due to changing non-functional requirements. As mentioned earlier, we propose that aspect oriented programming is a technique that can “fix” the Achilles heel of agile methods.

In the next section we will give an example of how aspect oriented programming can be used to implement non-functional requirements without major rework of the original system.

An example: Adding QoS Metrics to Web services

The example in this section describes the steps necessary to extend a web service with quality of service metrics monitoring (QoS monitoring). The example elucidates the main point of this paper, that by using aspects, new non-functional requirements can be introduced late in an agile development project. The need to extend web services with QoS metrics is selected as an example both because it is a likely scenario, and because it clearly demonstrates how non-functional requirements can be implemented using aspects. What makes the scenario likely is that enterprises starting to use the web service technology internally (intra enterprise use) will need to further define, and monitor their quality of service when starting to use web service technologies as a communication mean between enterprises (extra enterprise use), thus the need to add QoS metrics to web services.

Scenario

To illustrate how AOP will help in implementing non-functional requirements such as QoS metrics let's imagine a company that provides financial services such as mortgages and loans for cars. In this business it is essential to know your customers' credit worthiness. Credit worthiness is determined using the customers' credit history, income and other variables. Different financial services require different definitions and levels of credit worthiness. This information is used in determining whether to grant applications for both loans and mortgages. Since credit checking is an important part of this organisation's business, it is implemented as a web service that can be reused from all systems within the

organisation. Figure 1, below, shows how the interface to this credit checking service might look like implemented in Java.

```
public interface CreditCheckingServiceInterface
{
    public boolean hasPaymentRemarks(String name);
    public boolean hasCreditHistory(String name);
    public boolean checkCreditForAmount(String name, int amount);
}
```

Figure 1. The interface of the CreditCheckingService

The next step in the organisation's business plans could be to provide the credit checking service to external businesses, such as mobile phone operators and car leasing companies that also need efficient credit check processing. However, before using the credit checking from their systems, external businesses will require some form of quality guarantee. For example, a potential customer of the service would probably ask the following questions:

- How can it be ensured that the service paid for is reliable and running when needed?
- How can the organisation monitor that the performance is acceptable?

In short, the customers will require some form of agreement that states the intended quality of service. The agreement can include measurable limits for performance, cost, up time and other dimensions that affects the overall quality of the provided web service. For this example we use three QoS dimensions for web service processes as defined by Sheth et al. (2002): time, cost and reliability.

- *Time* is a measure of response time of the web service that is to be monitored. The response time is measured from request arrival to the completion of the request.
- *Cost* can be measured by either estimating an average cost for each service invocation, or by measuring the resources that are consumed to complete a request (such as processor time, cost of information storage etc).
- *Reliability* is a measure of technical failure rate, that is monitoring the reliability will discover how many times the service failed to deliver a response. Sheth et al. (2002) suggest that reliability is to be measured as a ratio of successful executions/scheduled executions.

The credit checking web service mentioned above is not built with QoS metrics in mind, since it was designed for internal use only. Using plan-oriented, non-agile methods, this new requirement might have been anticipated for when the service was first constructed. Using agile methods however, the new requirement needs to be added to the existing design. Adding QoS metrics to the existing service can be a major undertaking, since code that monitors the metrics need to be inserted in all parts of the service. Without a technique that helps implement cross-cutting, non-functional requirements such as QoS metrics, users of agile methods are running the risk of having to redesign a major part of the code. However, applying aspect oriented programming can reduce this risk. An example of how aspects can be applied in this case is described in the next section.

Applying Aspects

Let's look at how aspects could be applied in the described scenario. The three QoS dimensions time, cost and reliability define what is to be measured. Before implementing the actual metrics, it has to be decided where in the application code the dimensions should be measured. A basic approach would be to add code to register each metric in the beginning and end of each request, i.e. before and after each call to the web service. Without using aspects, this approach would require additional code that has to be inserted in *all* web service methods. However, using an aspect-oriented approach, adding QoS metrics to web services would only require the metrics *aspects* and their *join points* to be defined once, without any change to the original web service implementation.

To implement QoS metrics for the credit checking service, one aspect for each of the QoS dimension can be implemented. Thus, as an example we have implemented the aspects `PerformanceQoSAspect`, `CostQoSAspect` and `ReliabilityQoSAspect`.

Performance Aspect

The performance aspect is intended to measure the Time QoS dimension. Time can be measured by recording the request/method name, when the request arrived and when the response was sent. The implementation of this metric requires that two aspect join points are defined; one at the beginning of each method call and one at the end. These join points are defined within the AspectJ pointcut "timedMethods", see figure 2.

```

public aspect PerformanceQoSAspect
{
    Timer timer=new Timer();

    pointcut timedMethods() : (
        execution(public * CreditCheckingService.* (..)));

    before() : timedMethods()
    {
        // Start timing
    }

    after() : timedMethods()
    {
        // End timing
    }
}

```

Figure 2. Performance QoS aspect

Cost Aspect

Cost can be measured by recording the request/method name for each request. Using predefined cost for each type of request, the total cost can be calculated. The measurement of cost can be done by using a join point at the end of each web service method. The AspectJ example in figure 3 show how an aspect that logs each methods call can be implemented.

```

public aspect CostQoSAspect
{
    pointcut costMethods() : (
        execution(public * CreditCheckingService.* (..)));

    after() : costMethods()
    {
        // Log the cost of the executed method
    }
}

```

Figure 3. Cost QoS aspect

Reliability Aspect

Reliability can be measured by recording if the response of a request is a valid response or an error. In this case, a join point can be defined at the end of each method. The AspectJ implementation shown in figure 4 defines an aspect that logs every method call that ends with a non-application Exception.

```

public aspect ReliabilityQoSAspect
{
    pointcut reliabilityMethods() : (
        execution(public * CreditCheckingService.* (..)));

    after() throwing(Exception e): reliabilityMethods()
    {
        if(!(e instanceof ApplicationException))
        {
            // Log error
        }
    }
}

```

Figure 4. Reliability QoS aspect

The example given above can be extended with more QoS metrics. This example illustrates the main points in using aspects for the implementation of non-functional requirements.

Conclusion

In this article we propose that AOP could augment agile methods. By combining agile methods and AOP, the flexibility of agile methods can be maintained without risking major redesign when non-functional requirements change. The feasibility of the proposed solution has been demonstrated with a simple example written in AspectJ.

The example demonstrated that AOP could be a useful tool when an application needs to accommodate QoS metrics that have not been previously designed into the system. It also

shows that this can be easily achieved using just a few lines of code. In fact, the bigger the application, the more amount of time will be saved by using aspects.

AspectJ was used in the example, however, there are other ways to implement non-functional requirements in an “aspect oriented” way. It could be argued that by using a component technology such as Enterprise Java Beans (EJB), QoS metrics could be provided by the application server (e.g. through the use of method interceptors in the JBoss server (JBoss)). These QoS metrics, however, are not currently standardised, they would therefore be different for each component server. It would also require the application to be built as a component based-application from the start, which is often a lot more time-consuming and skill-intensive than using plain Java objects. Using design patterns such as the “proxy” pattern (Gamma et al., 1995) could also alleviate the need for using specific AOP technologies such as AspectJ. An example of this is provided by Filman et al. (2002). This approach, however, is considerably more time-consuming and therefore also more error-prone. The same goes for existing refactoring techniques (Fowler, 1999).

The proposed solution is applicable when non-functional requirements are largely unknown in the beginning of an agile software development project. If the requirements are known or predictable in the beginning of a project the best solution may not always be the combination of agile development processes and AOP. However, a situation where all the requirements are known up-front is not a likely scenario. This applies for non-functional requirements as well. Agile methods are very well equipped to handle changes in requirements, but due to their just-in-time design principles agile methods do not cope well with large changes in the architecture. Thus, we believe that development teams need an easy way to implement non-functional requirements in an agile way.

In this article, the only agile principles that specifically has been taken into consideration is simplicity, which has served as the starting point for our discussion about

how non-functional requirements can be handled late in agile software development projects. We suggest that AOP would be a simple solution to handle the weaknesses identified with this approach, but this that does not imply that AOP is simple to use in itself. We have not taken into consideration what effects the use of AOP may have on the other core principles of the agile methodology, such as communicating design, delivering early and continuously, and involving business people (Beck et al., 2001). This discussion is out of the scope of this paper, but Kircher et al. (2002) have made an interesting contribution on the impact of AOP on the practices of XP.

Further work

The presented example, the credit checking service, is realistic, but still hypothetical. Real-world examples would be of more value for assessing the results.

An interesting question is whether AOP also could prove useful for solving other “architecture breaking” problems. There are several indications that this could be the case. De Win et al. (2002) have shown how AspectJ can be used to help implement security features in an application. Filman et al. (2002) have also described how AOP can be used for inserting “ilities”, such as stability and reliability. These examples, however, do not prove that AOP can handle every possible new requirement, but we believe it shows that AOP is a valuable tool that supports the agile methodologies’ practice of evolutionary design.

References

AspectJ, www.aspectj.org. Accessed in April 2003.

Beck K., *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 2000.

Beck K., et al., *The Agile Manifesto*. www.agilemanifesto.org, 2001. Accessed in April 2003.

Boehm B., *Get Ready for Agile Methods, with Care*. IEEE Computer, 2002, Vol. 35 (1), pp. 64-69.

Cysneiros L., do Prado Leite J., *Non-Functional Requirements: From Elicitation to Modelling Languages*. International Conference on Software Engineering, 2002.

Elrad T., Filman, R., Bader A., *Aspect-oriented Programming an Introduction*. Communications of the ACM, 2001, Vol. 44 (10).

De Win B., Vanhaute B., De Decker B., *How Aspect-Oriented Programming Can Help to Build Secure Software*. Informatica, 2002, Vol. 26 (2), pp.141-149 (Belgian IT Journal).

Duclos F., Estublier J., Morat P., *Describing and Using Non Functional Aspects in Component Based Applications*. 1st International Conference on Aspect-Oriented Software Development, 2002.

Filman R., et al., *Inserting Ilities by Controlling Communications*. Communications of the ACM, 2002, Vol. 45 (1).

Fowler M., *The New Methodology*.

www.martinfowler.com/articles/newMethodology.html. 2003. Accessed in April 2003.

Fowler M., et al, Refactoring – *Improving the Design of Existing code*. Addison-Wesley, 1999.

Fremantle P., Weerawarana S., Khalaf R., *Enterprise Services*. Communications of the ACM, 2002, Vol. 45 (10).

Gamma E., Helm R., Johnson R., Vlissides J., *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

JBoss, www.jboss.org, Accessed in April 2003.

Jeffries R., Anderson A., Hendrickson C., *Extreme Programming Installed*. Addison-Wesley, 2001.

Kelter U., Monecke M., Schild M., *Do We Need “Agile” Software Development Tools?* Net.ObjectDays, 3rd united GI conference on "Object-Oriented Programming for the Net world", 2002.

Kircher M., Jain P., Corsaro A., *XP + AOP = Better Software?* Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, 2002.

Lindvall M., et al. *Empirical Findings in Agile Methods*. www.cebase.org, 2002. Accessed in March 2003.

Moriera A., Araújo J., Brito I., *Crosscutting Quality Attributes for Requirements Engineering*. 1st International Conference on Aspect-Oriented Software Development, 2002.

Sheth A., Cardoso J., Miller J., Kochut K., and Kang M., *QoS for Service-Oriented Middleware*. Proceedings of the Conference on Systemics, Cybernetics and Informatics, 2002.

Wendorff P., *An Essential Distinction of Agile Software Development Processes Based on Systems Thinking in Software Engineering Management*. Third International Conference on eXtreme Programming and Agile Processes in Software Engineering, 2002.